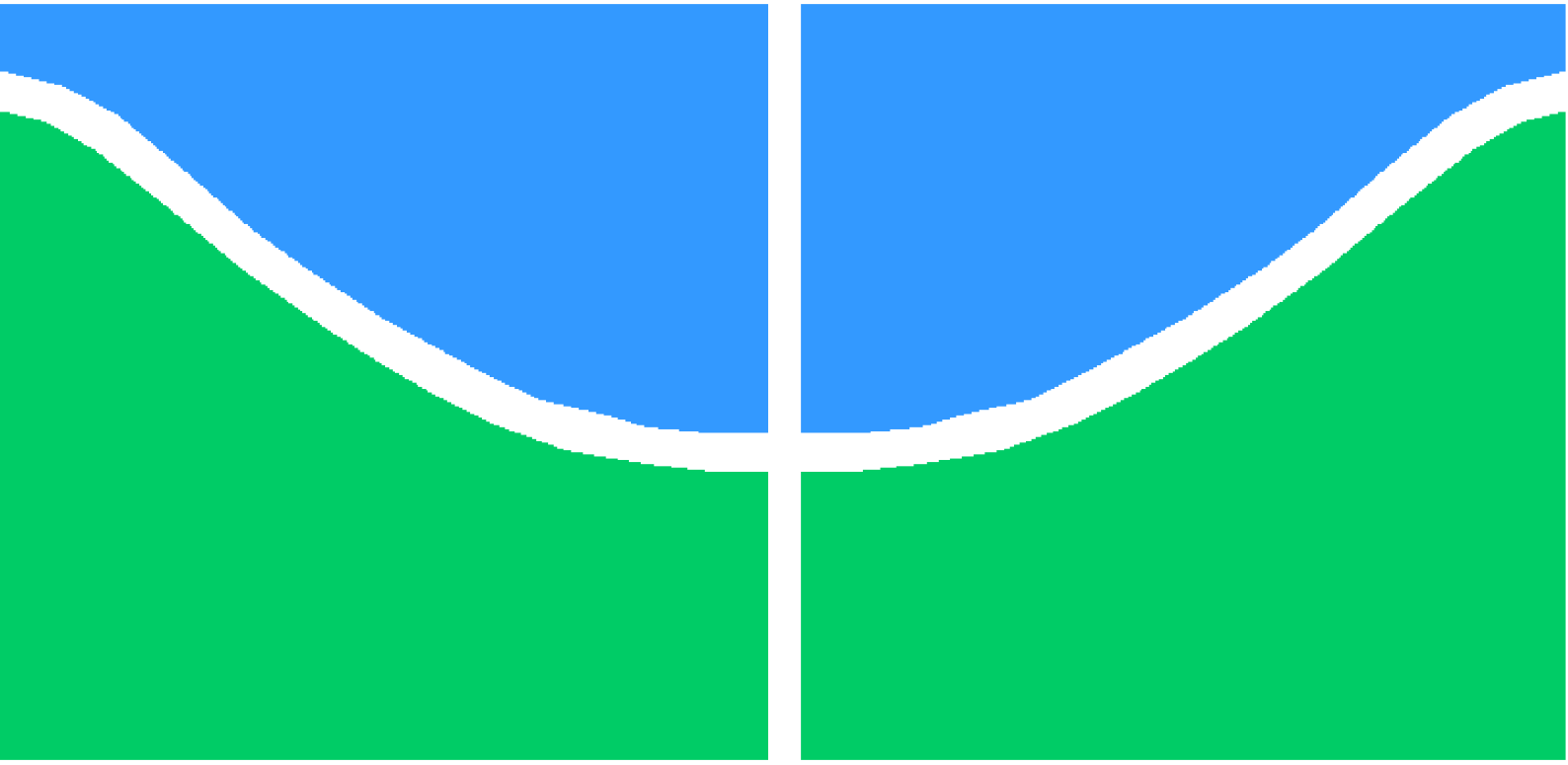

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Engenharia de software

Identificação de erros semânticos em código-fonte oriundos da representação numérica em computadores e em matemática

Autor: Julliana do Couto Almeida

Orientador: Dr. Edson Alves da Costa Júnior

Brasília, DF

2019

Julliana do Couto Almeida

Identificação de erros semânticos em código-fonte oriundos da representação numérica em computadores e em matemática

Monografia submetida ao curso de graduação em Engenharia de software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de software.

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Orientador: Dr. Edson Alves da Costa Júnior

Brasília, DF

2019

Julliana do Couto Almeida

Identificação de erros semânticos em código-fonte oriundos da representação numérica em computadores e em matemática/ Julliana do Couto Almeida. – Brasília, DF, 2019-

58 p. : il. (algumas color.) ; 30 cm.

Orientador: Dr. Edson Alves da Costa Júnior

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2019.

1. semântico. 2. Problema. I. Dr. Edson Alves da Costa Júnior . II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Identificação de erros semânticos em código-fonte oriundos da representação numérica em computadores e em matemática

CDU 681.3.41

Julliana do Couto Almeida

Identificação de erros semânticos em código-fonte oriundos da representação numérica em computadores e em matemática

Monografia submetida ao curso de graduação em Engenharia de software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de software.

Trabalho aprovado. Brasília, DF, 13 de dezembro de 2019:

Dr. Edson Alves da Costa Júnior
Orientador

Dr. John Lennon Cardoso Gardenghi
Convidado 1

Me. Daniel Saad Nogueira Nunes
Convidado 2

Brasília, DF
2019

Agradecimentos

Agradeço aos meus pais, Weber Jaime de Almeida e Ozana Aparecida do Couto pelo apoio incondicional, dedicação e os valores que me ensinaram. Agradeço aos meus irmãos Felipe Duerno do Couto Almeida e Jéssica Karine do Couto Almeida, por sempre me incentivar e me apoiar em todas as decisões tomadas.

Agradeço, ainda, à todos meus amigos, em especial à Joyce da Costa Santos por estar comigo na biblioteca semestre a semestre se fazendo valer o sentido da palavra amizade. Agradeço também à todas as mulheres que escolheram atuar na área de tecnologia mostrando a representatividade feminina, e incentivando outras mulheres a se sentirem capazes de ocupar esse espaço.

Por fim, meu agradecimento especial à meu orietador prof. Dr. Edson Alves da Costa Júnior, por ser uma referência de incentivo à educação e uma pessoa excepcional. Obrigada por valorizar nós alunos, mostrando com seu próprio trabalho o que realmente faz diferença para se ter bons resultados. Além disso, por todo apoio e auxílio durante a execução deste trabalho.

"Em certos os casos, quanto mais nobre o gênio, menos nobre o destino.
Um pequeno gênio ganha fama,
um grande gênio ganha descrédito,
um gênio ainda maior ganha desprezo;
um deus ganha crucificação."
(Fernando Pessoa)

Resumo

Produzir código gera erros que fazem parte do processo de aprendizagem. O problema disso é que muitas vezes eles não apresentam indícios visuais, o que significa que não geram erros pelo compilador. Neste caso, o erro é dito semântico, ele ocorre quando o desenvolvedor escreve um bloco de código que não condiz com a teoria naquele contexto. O erro semântico acontece porque as propriedades contidas nas operações do computador não reproduzem o que acontece na matemática. A associação, por exemplo, é propriedade válida dos inteiros que não acontece em todas as entradas em código.

O objetivo deste trabalho, é apresentar problemas de erros semânticos de *overflow* em inteiros, *overflow* em ponto flutuantee, adicionalmente, classificar o operador % de acordo com seu retorno nas linguagens mais populares de 2019. Para os problemas de *overflow* houveram exemplos onde foi possível demonstrar matematicamente a taxa de erro esperada. E para todos os casos de *overflow* foram realizados experimentos, onde comparo os códigos com e sem overflow utilizando valores randômicos para as respectivas variáveis. E então, gero a taxa de erro para $10, 10^2, 10^3, 10^4, 10^5$ e 10^6 casos de testes, ilustradas no gráfico de cada problema.

Palavras-chaves: Erros semânticos, *overflow*.

Abstract

Producing code causes errors that are part of the learning process. The problem of this is that they often do not show visual cues, which means that they do not generates errors by the compiler. In this case, the error is said to be semantic, it occurs when the developer writes a block of code that does not fit the theory in that context. The purpose of this paper is to present code examples where there are these errors followed by their respective solution.The examples chosen for this work were: arithmetic mean, minimum common multiple and remainder of the division, zero division of numbers floating point and associativity of three elements in the sum and multiplication.

The first two examples reflect data invalid in overflow, to verify the error rate was plotted a comparative chart, already for the third case, there is a same operator that generates different results depending of the language it has been implemented, the proposed solution for this case is a table comparing them by classifying them. The results were 50% chance of the function of the mean return an invalid value while in the case of mmc were 31%.

The last two problems work with floating point overflow, the fourth problem being referring to division by zero and their respective values congruent to it. In this problem it was verified than computers that use 32 textit bit archiving, this value is congruent to zero when implemented in code, such as body mass index ($IMC = fracmh^2$) when $h = 16$. Lastly, and Not least, the associativity problem in addition and multiplication of three elements. That problem reports the sum of three decimal numbers results in different values depending on the order in which were willing. This same scenario happens in three floating point numbers when multiplied with 38.8% chance of invalid results, while in total the chance is 36.1%.

Key-words: semantic errors, overflow.

Lista de ilustrações

Lista de tabelas

Lista de símbolos

$\mathbb{N}$	Conjunto dos números naturais
$\mathbb{Z}$	Conjunto dos números inteiros
$a \in A$	Elemento a pertence ao conjunto A
$a \nmid b$	a não divide b
$a \mid b$	a divide b

Sumário

Introdução

Entender exatamente o que nos foi dito é um desafio. Especialmente porque na linguagem natural existem termos lexicais com mais de um sentido, fato conhecido como indeterminação linguística (ROSSA et al., 2001). O estudo dessa indeterminação bem como o sentido das palavras e a interpretação das sentenças é conhecida por semântica, enquanto o estudo responsável pela concordância dos elementos de uma frase, ordem de uma sentença é conhecida por sintaxe.

De maneira análoga, quando um programa apresenta um erro durante a compilação, provavelmente é um erro de sintaxe; no entanto, ao executar um código e ele, apesar de compilar corretamente, apresentar um erro, ou seja, o programa retorna resultados diferentes dos esperados, a chance maior é que se trata de um erro semântico. Estes são os erros que serão abordados nesse trabalho.

Muitas vezes ao lidar com a arquitetura de computadores assume-se que são realizadas operações no conjunto dos números inteiros, todavia nem todas as propriedades dos inteiros estão presentes. Isto acontece porque a arquitetura de computadores é baseada em conjunto com suas próprias propriedades, chamado de anel finito. Diante disso, esse trabalho investiga a seguinte questão: *Como as diferenças entre o conjunto dos números inteiros $\mathbb{Z}$ e o anel finito $\mathbb{Z}_n$ impactam a escrita do código-fonte e podem levar a erros semânticos?*

Produzir código usando variáveis do tipo `int` não é suficiente para resolver todos os problemas do dia-a-dia. Utiliza-se, então, ponto flutuante, que são valores decimais compostos pelo *bit* de sinal, a mantissa e o expoente. Assim como no contexto dos números inteiros, o código em ponto flutuante está sujeito a erros semânticos causados pelo *overflow*.

Objetivos

O objetivo geral é identificar trechos de código que estão sintaticamente corretos, mas possuem erros de execução oriundas da diferença semântica do con-

junto dos inteiros e anéis finitos, e também dispostas em pontos flutuantes e propôr escritas alternativas que eliminem tais erros. Os objetivos específicos são:

- elencar alternativas de código para evitar os erros semânticos apresentados;
- apresentar exemplos em código aberto referentes aos problemas propostos;
- analisar a porcentagem de erros ao comparar o bloco de código com *overflow* em relação à solução proposta;
- classificar as linguagens de programação quando há divergências nos resultdos para o mesmo conceito (resto da divisão).

Estrutura do Trabalho

Este trabalho está dividido em 5 capítulos. O embasamento teórico se encontra no Capítulo 1, o qual esclarece e define termos usados ao longo deste trabalho. A seguir, o Capítulo 2 descreve o desenvolvimento do presente trabalho. O Capítulo 3 apresenta a descrição de cada problema, a seguir no Capítulo 2.2.2 é apresenta suas respectivas soluções. Por fim, o Capítulo 4 aborda as considerações finais.

1 Fundamentação Teórica

Um motivo comum para resultados inesperados na execução de um código é o *overflow*. O *overflow* ocorre quando a quantidade de *bits* reservada para representar um número não é suficiente. Outro motivo é o fato de que linguagens diferentes implementam funcionalidades com o mesmo nome mas resultam em ações distintas. Essas diferenças podem levar a erros, caso o desenvolvedor as desconheça. Neste capítulo serão apresentados os conceitos de *overflow* e definições relativas à Teoria dos Números utilizadas nas implementações de operadores aritméticos com os operadores de divisão e resto da divisão.

1.1 Teoria dos Números

A Teoria dos Números é a área da matemática que se dedica a estudar as propriedades dos números inteiros. Ela desenvolveu estudos importantes para o avanço da ciência. O Teorema de Euclides, por exemplo, é um dos dos principais resultados da área, que demonstra a existência de infinitos números primos (NEVES, 2011).

O conjunto dos números inteiros é formado por números positivos, negativos e zero, representado por $\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$. A notação $\mathbb{Z}^*$, por sua vez, representa o conjunto $\mathbb{Z}$ excluindo-se o zero.

Dois conceitos são fundamentais para a matemática como um todo, bem como para o entendimento desse trabalho: as operações de adição e multiplicação. Elas auxiliaram as civilizações em suas trocas comerciais e em outras situações do seu cotidiano. Inicialmente estas operações eram efetuadas em tabelas, porém, o cálculo envolvendo grandes números era demorado e pouco efetivo, o que levou ao desenvolvimento de técnicas que permitissem realizar tais operações com maior eficiência (AROEIRA, 2018).

Em termos formais, a adição nos inteiros é uma operação binária $+ : \mathbb{Z}\times\mathbb{Z} \rightarrow$

$\mathbb{Z}$ tal que para $(x, y) \in \mathbb{Z} \times \mathbb{Z}$

$$\begin{cases} x + 0 = x \\ x + y' = (x + y)', & \text{se } y > 0 \\ x + y'' = (x + y)'', & \text{se } y < 0, \end{cases} \tag{1.1}$$

onde n' é o sucessor de n ($n' = n + 1$) e n'' é o antecessor de n ($n'' = n - 1$).

De forma semelhante, a multiplicação nos inteiros é uma operação binária $\times := \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ tal que para $(x, y) \in \mathbb{Z} \times \mathbb{Z}$

$$\begin{cases} x \times 1 = x \\ x \times y' = x \times y + x, & \text{se } y > 0 \\ x \times y'' = x \times y + (-x), & \text{se } y < 0, \end{cases} \tag{1.2}$$

onde $(-x)$ é o simétrico de x, isto é, $x + (-x) = 0$.

Na prática, estas operações são realizadas por meio de tabuadas ou com auxilio de ferramentas mecânicas (por exemplo, o ábaco) ou eletrônicas (calculadoras, computadores, celulares, etc).

As definições de adição e multiplicação acima permitem a demonstração das propriedades listadas na Proposição 1.

Proposição 1 (Propriedades da Adição e Multiplicação) Sejam $a, b, c \in \mathbb{Z}$. Então

i. $(a + b) + c = a + (b + c)$ e $(a \times b) \times c = a \times (b \times c)$ (associatividade)

ii. $a + b = b + a$ e $a \times b = b \times a$ (comutatividade)

iii. Existem $0, 1 \in \mathbb{Z}$ tais que $a + 0 = a$ e $a \times 1 = a$ (elemento neutro e identidade)

iv. Existe $(-a) \in \mathbb{Z}$ tal que $a + (-a) = 0$ (elemento simétrico)

As propriedades de multiplicação possibilitaram os estudos de divisibilidade, conforme apresentados na Definição 1. Este conceito deu início aos teoremas iniciais na Teoria dos Números, como por exemplo o Teorema 2, que aborda a divisão euclidiana.

Definição 1 (Divisibilidade) Sejam a e b inteiros. Dizemos que a divide b se existe um inteiro c tal que $b = ac$. Se a divide b, b é chamado múltiplo de a e a é chamado divisor de b. Se a divide b usa-se a notação $a \mid b$, caso contrário, usa-se $a \nmid b$.

O Teorema 2 estabelece uma relação entre dois inteiros a e b por meio de um quociente q e um resto r e que expande a ideia de divisibilidade.

Teorema 2 (Divisão de Euclides) Sejam a e b inteiros, com $b \neq 0$. Então existem únicos $q, r \in \mathbb{Z}$ tais que

$$a = bq + r$$

com $0 \leqslant r < |b|$.

Outro cenário possível é a a divisão de a por b ser exata, ou seja neste caso não haverá resto (r=0) e vale a operação $a = bc$.

Definição 2 (Maior Divisor Comum(MDC)) Dados dois números inteiros a e b, o maior divisor comum de a e b é o maior número inteiro que divide ambos. O MDC sempre está definido, pois 1 divide qualquer inteiro, isto é, $1 \mid x$, $\forall x \in \mathbb{Z}$.

Definição 3 (Menor Divisor Comum(MMC)) Dados dois números inteiros $a, b \in \mathbb{N}$, o menor divisor comum deles é o menor inteiro positivo que divide a e b.

A partir do Teorema de Euclides construiu-se uma série de conceitos mais elaborados, como, por exemplo, a congruência entre dois números inteiros.

Definição 4 (Congruência) Dados a e b inteiros, eles são ditos congruentes módulo M se M divide a diferença de a e b. Notação: $a \equiv b \pmod{M}$.

Isto significa que a diferença $a - b$ pode ser igualada a um produto km, para algum $k \in \mathbb{Z}$. Mas a e b, por serem números inteiros, podem ser reescritos como

$qm + r$, isto é, $a = q_a m + r_a$, $0 \leq r_a < m$ e $b = q_b m + r_b$, $0 \leq r_b < m$. Estas igualdades levam a:

$$km = (a - b) = (q_a m + r_a) - (q_b m + r_b).$$

Isolando o termo em comum m segue que:

$$mk = m(q_a - q_b) + (r_a - r_b).$$

Ao resolver essa equação, a igualdade deve se manter, o que significa que o termo $(r_a - r_b)$ deve ser igual a 0. Concluimos então que os restos são devem ser iguais, isto é,

$$r_a = r_b.$$

Assim, uma definição alternativa para a congruência seria: a e b são congruentes módulo m se ambos deixarem mesmo resto quando divididos por m.

A classe de equivalência de a módulo m é representada por $[a]$ ou simplemente $\bar{a}$, e abrange o conjunto de números inteiros congruentes a a módulo m.

Definição 5 (Classe de Equivalência) Uma classe de equivalência $[a]$ é o conjunto de todos os inteiros congruentes a a módulo m, isto é, $[a] = \{b \in \mathbb{Z} \mid a \equiv b \ (mod\ m)\}$. Ou seja, uma classe de equivalência é induzida pela congruência, é uma relação de equivalência entre os valores do conjunto.

Por outro lado se b não pertence à classe de equivalência de a, ou seja, $b \notin \bar{a}$, certamente a e b não tem os mesmos restos. Entendemos então, que para cada valor de resto, existe uma classe de equivalência, de forma que podem existir apenas m classes de equivalência distintas. Caso dois valores retorne tenham mesmo resto, isso significa que eles pertencem a mesma classe de equivalência.

Definição 6 (Sistema de Resíduos) Um sistema de resíduos é um conjunto de números inteiros $S_m = \{s_1, s_2, \cdots, s_k\}$, tal que cada elemento s_i pertence à uma classe distinta. Isto equivale a $s_i \not\equiv s_j (\text{mod m})$, se $i \neq j$.

Por exemplo, para $m = 5$, o conjunto $S_m = \{20, 43\}$ é um sistema de resíduos, isso porque o menor representante positivo da classe vinte [20] é zero enquanto o menor representante positivo da classe de equivalência do [43] é 3, um número diferente de zero. Porém, o conjunto $S_2 = \{10, 20\}$, não é um sistema de resíduos pois, para o mesmo $m = 5$, tanto o elemento 10 quanto 20, ao serem divididos pelo m, resultam o mesmo resíduo: zero.

Definição 7 (Sistema Completo de Resíduos) Um sistema completo de resíduos S_m que é um sistema de resíduos com $|s_m| = m$. Ou seja, $\forall b \in \mathbb{Z}$, existe $s_j \in S$ tal que $b \equiv s_i \pmod{m}$.

Por exemplo, para $m = 5$ e $S_m = \{-2, 15, 1, 1, 19, -8\}$, as classes equivalêntes são: $-2 \in [3]$, $15 \in [0]$, $1 \in [1]$, $19 \in [4]$, $-8 \in [2]$. Isso significa que no sistema completo de resíduos todas as classes de equivalência devem estar representadas, mantendo a incongruência entre todos os seus elementos.

Para $m > 1$, o conjunto $S_m = \{0, 1, 2, \cdots, m-1\}$ S_m é denominado sistema de restos euclidiano, conhecido também por sistema de resíduos canônico. Este é o sistema é usado em matemática no ensino fundamental e médio. Todavia, há outros sistemas completos de resíduos com aplicações práticas, como sistema de menor resto:

$$\left\{\left\lceil\frac{-m}{2}\right\rceil, \left\lceil\frac{-m}{2}\right\rceil + 1, \cdots, 1, 2, 3, \cdots, \left\lfloor\frac{m}{2}\right\rfloor\right\}$$

O sistema de menor resto com $m = 7$ elementos seria dado por:

$$R_7 = \{-3, -2, -1, 0, 1, 2, 3\}$$

Definição 8 (Sistema Reduzido de Resíduos) Para $m > 1$, um sistema reduzido de resíduos é representado pelo conjunto $SRR(m) = \{s_1, s_2, \cdots, s_k\}$ tal que sejam válidas as seguintes condições:

i. Para todo $b \in \mathbb{Z}$ com $(b, m) = 1$ existe $s_i \in SRR(m)$ tal que $b \equiv s_i \pmod{m}$

ii. $(s_i, s_j) = 1$, se $i \neq j$

Isso significa que um sistema reduzido de resíduos é composto pelos valores co-primos de m. Por exemplo para $m = 5$, $SRR(5) = \{1, 2, 3, 4\}$ e, para $m = 12$, $SRR(12) = \{1, 5, 7, 11\}$, são dois exemplos de sistemas reduzidos de resíduos.

1.2 *Overflow*

Em linguagens de programação, uma variável é capaz de armazenar um valor de acordo com o número de *bits* disponível para o tipo que ela foi declarada. Caso um valor ultrapasse esse número de *bits* reservado, ocorrerá um transbordamento de capacidade, conhecido como *overflow* (FLOYD, 2007).

A linguagem C reserva 4 *bytes* de armazenamento para variáveis do tipo *long* nas platarformas Unix, Windows, exceto em plataformas Unix de 64-*bits*, as quais contam com espaço de 8 *bytes*, conforme ilustra a Tabela 1 (RITCHIE, 1996).

.

Tabela 1 – Valor armazenado em variáveis de 4 *bytes* e 8 *bytes*

bytes	**Valores limites**
4	-2147483648 até +2147483647
8	-9223372036854775808 até +9223372036854775807

Sabendo desses valores é possível *hackear* dados definindo informações em *bits* memória com tamanho insuficiente para armazená-los, fazendo com que haja sobrescrita das posições de memória adjacentes ao espaço reservado, alterando seus conteúdos (GUIDETTI et al., 2005). Nesse caso é recomendado testar dados a fim de encontrar possíveis *overflows* para evitar resultados inválidos.

1.3 Problemas vinculados ao *overflow*

Este trabalho apresenta exemplos de quatro aplicações onde *overflows* levam a resultados inválidos: média aritmética, cálculo do mínimo múltiplo comum de dois

inteiros, divisão por zero e associatividade na soma e produto, ver Capítulo 3.

1.4 Anéis

Definição 9 (Anel) Anel é uma estrutura algébrica composta por um conjunto não vazio A provido de um par de operações: uma adição $\oplus$ e uma multiplicação $\odot$ (EVARISTO; PERDIGÃO, 2002), dotadas das seguintes propriedades:

i. a adição de um anel A é associativa, comutativa e possui um elemento neutro $0_A \in A$ tal que a + 0_A = a.

ii. na multiplicação, as propriedades ficam restritas a existência do elemento identidade 1_A.

iii. simetria em anéis $\forall a \in A \; \exists b \in A : a + b = 0$

O anel tem o elemento inverso de a se existir $b \in A$ tal que $a \odot b = 1$. Nesse caso, a é dito invertível em A e b é o inverso multiplicativo de a. Um anel $(A, \oplus, \otimes)$ cujo conjunto A é finito é denominado anel finito (GONCALVES, 2002). Vale ressaltar que um anel pode apresentar divisores de zero, os quais são definidos na Definição 10.

Definição 10 (Divisor de zero) Dado um anel $(A, \oplus, \otimes)$, o elemento a é dito divisível por zero se existir um elemento b tal que $b \in A$, $b \neq 0$, tal que $a \otimes b = 0$.

1.5 Classificação de Anéis

Esta seção apresenta quatro tipos de anéis usados no desenvolvimento do trabalho: anéis comutativos, anéis com unidade, anéis comutativos com unidade e anéis de integridade.

Definição 11 (Anel Comutativo com Unidade) Seja $(A, \oplus, \otimes)$ um anel que contenha a propriedade comutativa para a multiplicação, ou seja, para quaisquer

$a, b \in A$, vale a igualdade $a \otimes b = b \otimes a$. Então A é um anel comutativo. Se este anel A conter o elemento neutro da multiplicação $1_A \in A$, assim $a \cdot 1_A = 1_A \cdot a = a$, ele é dito anel com unidade. No caso de apresentar as duas propriedades este anel é dito anel comutativo com unidade.

A estrutura $(A, \oplus, \otimes)$ é um anel que se classifica nas três categorias. Isso acontece o conjunto dos números inteiros possui adição e multiplicação comutativa e elemento neutro de unicidade (IEZZI, 2003).

1.5.1 Domínio de integridade

Um anel é conhecido como anel de integridade, ou domínio de integridade, quando a multiplicação apresenta as seguintes propriedades:

1. elemento neutro;
2. comutatividade;
3. ausência de divisores de zero, isto é, $\forall x, y \in A, x \otimes y = 0$ implica em $x = 0$ ou $y = 0$.

Considerando o anel $(\mathbb{Z}, \oplus, \otimes)$, que é um domínio de integridade, a equação: $(x-1)(y+2) = 0$ possui apenas duas soluções: $x-1 = 0$ ou $y+2 = 0$. Como se tratam de soluções únicas dizemos que a equação é possível e determinada. Um exemplo de equações com infinitas soluções é o anel usado nos computadores que, apesar de ter algumas características do conjunto dos inteiros, pode possuir divisores de zero não triviais. Outro exemplo é o anel das matrizes $(M_{2\times 2}(\mathbb{R}), \oplus, \otimes)$. A equação abaixo possui infinitas soluções:

$$\left(X - \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \right) \left(Y - \begin{bmatrix} 0 & 0 \\ -1 & 1 \end{bmatrix} \right) = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Exemplos de soluções são:

$$X = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, \; Y = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

$$X = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}, \; Y = \begin{bmatrix} 0 & 0 \\ -1 & 1 \end{bmatrix}$$

$$X = \begin{bmatrix} -5 & 1 \\ 0 & 0 \end{bmatrix}, \; Y = \begin{bmatrix} 1 & 0 \\ 5 & 0 \end{bmatrix}$$

A multiplicidade de soluções acontece porque o anel $(M_{2\times 2}(\mathbb{R}), \oplus, \otimes)$ tem divisores de zero.

2 Metodologia

Este capítulo apresenta a visão geral do trabalho e suas respectivas etapas. A seção a seguir detalha o fluxo de trabalho para as etapas de documentação, implementação e análise de resultados. Em seguida estão descritas as fontes bibliográficas e as ferramentas de trabalho utilizadas.

2.1 Visão Geral do Trabalho

Este trabalho tem caráter exploratório por proporcionar inovação no agrupamento dos problemas. Para tal efeito foram implementados algoritmos desprovidos de erros semânticos, tema carente em refencial bibliográfico contemporâneo.(RAUPP; BEUREN, 2006) Tal execução foi construída mediante a um catálogo com soluções para cinco dos problemas requisitados na engenharia de *software* conforme Tabela 2.

.

Tabela 2 – Problemas discutidos no trabalho

Problemas	**Descrição**
1	Média Aritmética
2	Mínimo Múltiplo Comum
3	Menor Resto
4	Divisão por zero
5	Associatividade na adição e multiplicação

2.2 Fluxo de Trabalho

O andamento do trabalho foi estabelecido por meio de reuniões semanais com o orientador, as quais tiveram duração de aproximadamente uma hora. Nestas foram discutidos quais seriam os principais problemas de incoerência semântica,

suas respectivas implementações e posteriormente discutida a organização da escrita do trabalho. A execução do trabalho guiou-se pelo processo ilustrado na Figura 1.

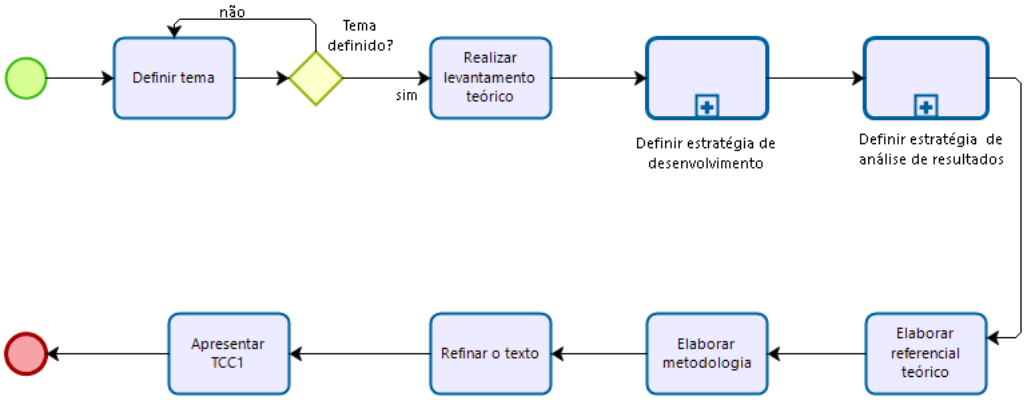

Figura 1 – Macro processo de atividades do trabalho de conclusão de curso 1

Conforme apresentado no diagrama da Figura 1, o primeiro passo foi definir o tema em conjunto ao orientador, dado o interesse de abordar matemática neste trabalho. Por isso, foi pensado um tema que estivesse presente na Engenharia de Software, tivesse cunho educacional e fosse voltado para a matemática.

Em seguida foram definidas as estrátegias de desenvolvimento e análise de resultados, que são macro atividades dispostas no fluxograma da Figura 1 e estão detalhadas nos fluxos representadas pelas Figuras 2 e 4. Em sequência, foram elaborados o referencial teórico, e metodologia. Por fim, o texto foi refinado para defesa do TCC1 para a banca.

O TCC2 iniciou-se com a atualização do referencial teórico a fim de consolidar conceitos matmáticos usados no Capítulo 3. Em seguida foram feitos experimentos e discutidos os problemas referente à divisão por zero e associatividade. Em paralelo foram feitas as correções do Trabalho de Conclusão de Curso 1 e por último foi feita demonstrações matemáticas para os problemas 3.1, 3.4 e 3.5.

2.2.1 Estrátegia Para Desenvolvimento

Nas reuniões iniciais foi discutid como implementar os problemas definidos e organização da escrita do trabalho. Os problemas inicialmente selecionados foram recomendados pelo orientador com base em experiências de códigos incorretos em sala de aula e sua respectiva execução foi guiada pelo processo.

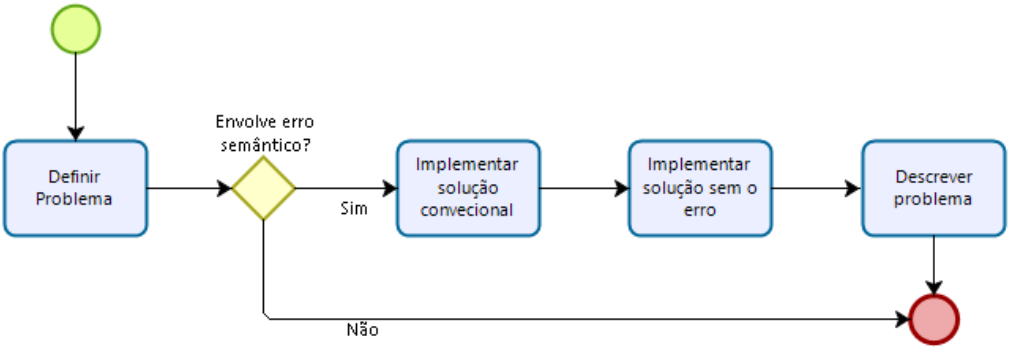

Figura 2 – Fluxograma de atividades do problema

Foram implementados os problemas em dois contextos: o primeiro foi por força bruta, sem examinar a fundo todos os possíveis casos enquanto a segunda implementação foi realizada de maneira completa, cobrindo possíveis *overflow* e incoerências semânticas. Com base nesses códigos, foram realizados testes comparativos para encontrar a taxa de erro ao testar valores randômicos para cada problema.

Para o problema 3.1 o experimento gera 10 pares (a, b) aleatórios para calcular a média entre eles e comparar o resultado com *overflow* e sem *overflow*, em seguida calcula a taxa de erro dos casos testados. Essa operação se repete para 10^2, 10^3, 10^4, 10^5, 10^6 casos de teste. De maneira análoga no Problema 3.2, o experimento gera os mesmos casos de testes para calcular o MMC.

Para o problema 3 foram feito testes em diversas linguagens a fim de analisar seu comportamento em cada uma delas, conforme apresentado na Tabela 3. Os avanços foram atualizados no Zenhub, que é um plugin adicionado ao navegador que analiza a produtividade do seu projeto do Github, como mostra a Figura 3. Essa ferramenta funciona como *kanban* ou *Just in time* que, é um sistema de

administração de produção.

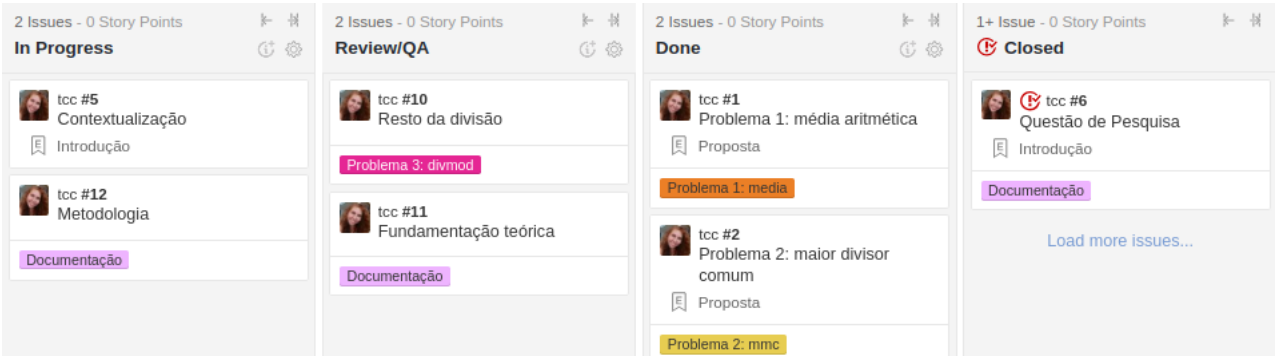

Figura 3 – Zenhub

Após isso, validou-se o estudo experimental na fundamentação teórica, no capítulo 2.2.2. Por fim, foi realizado um cronograma com a lista de tarefas pendentes para conclusão deste trabalho.

2.2.2 Estratégia Para Análise de Resultados

A montagem de resultados deste trabalho foi orientada à abordagem *bottom up*, por desenvolver primeiramente os códigos e depois agregar valor em documentação como mostra as Figuras 2 e 4 (DRUGAN, 2014). Esta metodologia é utilizada para aumentar a produtividade e engamento nas grandes empresas mesmo com o questionamento se esta técnica funciona sem recursos em maior escala de planejamento (LIEDL, 2011).

A execução de análise e resultados deste trabalho iniciou com a definição e validação da solução, escolhendo a melhor estratégia para mostrar os resultado do problema definido. Em seguida foram comparados resultados dos códigos produzidos com presença e ausência de *overflow*, com casos de testes randômicos para os problemas de média 3.1 e mínimo múltiplo comum 3.2. Já para o problema de resto da divisão 3.3, a métrica utilizada foi o número de linguagens que geram resultados diferentes para o mesmo operador. Em sequência o gráfico foi plotado e foi redigida uma redação descrevendo as soluções.

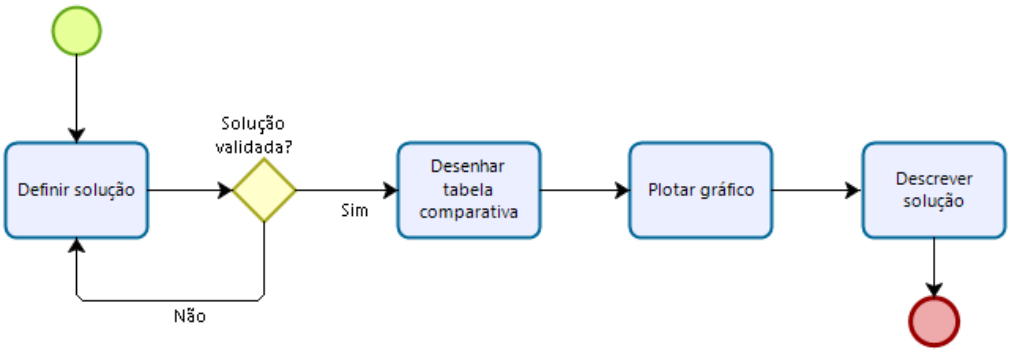

Figura 4 – Fluxograma de atividades da solução

2.3 Fontes Bibliográficas

As fontes bibliográficas para embasar o trabalho em teorias consolidadas foram: a Biblioteca Central da Universidade de Brasília (BCE) e Google Acadêmico[1]. A busca inicial de livros foi para suprir o *deficit* de conhecimento em álgebra estudado o curso de engenharia de *software* e também para reforçar os conceitos de Teoria dos números. Enquanto os outros temas como *overflow* e os conceitos pertinente aos problemas foram embasados em artigos encontrados pelo Google acadêmico.

2.4 Ferramentas de Trabalho

O computador utilizado para confecção de todo o trabalho foi um *notebook* da marca *Dell* modelo 14R 5437-A20, equipado com o processador Core i7- 4500U e com 7,7 GiB de memória, usando sistema de 64-*bits*. Os *softwares* utilizados para produção deste trabalho quanto a implementação, organização e escrita foram:

Os textos, algoritmos e gráficos foram feitos utilizando o ambiente de desenvolvimento integrado (IDE) *Visual Studio Code*, a qual foi escolhida por ser uma ferramenta com boa de usabilidade, contar com diversos *plugins* para diversas linguagens inclusive o LaTeX. Este é um sistema de preparação de documentos que faci-

[1] https://scholar.google.com.br/

Tabela 3 – Lista de ferramentas

Ferramenta	**Versão**	**Suporte**
Bizagi Modeler	3.4.1	Modelagem
Git	v.2.7.4	Controle de versão
GitHub	1.0.13	Hospedagem do código-fonte
Zenhub	v.2.38.101	*Kanban*.
LaTeX	v.6.2.1	Escrita/ formatação
Linux Ubuntu	v.16.04	Sistema Operacional
Visual Studio Code	v.1.33.1	Escrita de código e documentação
C	v.5.4.0	Desenvolvimento
C++	v.5.4.0	Desenvolvimento
C#	v .4.0	Desenvolvimento
Fortran	v.5.4.0	Desenvolvimento
Java	v.1.7.	Desenvolvimento
Javascript	v.1.7.10	Desenvolvimento
Perl	v.5.28.1	Desenvolvimento
PHP	v.7.3.7	Desenvolvimento
Python	v.3.4	Desenvolvimento
Ruby	v.2.6.0	Desenvolvimento
Visual basic.net	v.15.8	Desenvolvimento

lita na estrutura do trabalho, referencial bibliográfico e facilita a escrita de equações matemáticas utilizando comando pré-definidos e criação de tabelas.(LAMPORT, 1994)

O *Visual Studio Code* também foi usado para fazer implementações, sendo elas em sua maior parte utilizando a lingugaem C++ (versão 5.4.0) conhecida por dispor da *standard template library*(STL) que traz uma série de implementações de estruturas de dados codificadas em seu código-fonte. As demais linguagens estão descritas quanto a versão e compilador na tabela 3.

Para modelar os processos descritos neste capítulo foi utilizada a ferramenta *Bizagi* versão 3.4.1 [2] que descreve o passo-a-passo do desenvolvimento do trabalho. Para monitorar cada etapa usamos uma plataforma para o controle de versão neste caso o *Github*[3]. Trata-se de uma ferramenta gratuita que possibilita criar um repositório privado. Com isso foi possível adicionar o orientador como colaborador e seguir o kanban para planejar e monitorar a execução do trabalho pelo plugin *zenhub*.

[2] https://www.bizagi.com
[3] github.com

3 Resultados

Nesse capítulo serão apresentados exemplos práticos nos quais podem ocorrer erros semânticos em código. Isso significa que apesar de compilar e rodar, o código pode produzir resultados que não fazem sentido no contexto do problema.

Este trabalho evidencia, em cada problema, os casos em que ocorrem erros, o porquê do erro e apresenta um experimento comparando os blocos de código com presença e ausência de *overflow* gerados com casos de testes randômicos, inicialmente com 10 pares para os casos de testes, aumentando de $10k$ sendo k a quantidade de testes utilizado anteriormente no experimento, chegando até um milhão de casos de testes.

Este trabalho mostra aplicações no conjunto dos números inteiros e em ponto flutuante, ou seja implementações em int e $float$. Por exemplo ao calcular a associatividade entre três números, ela se mantém, porém existem casos em código que isto não acontece porque o computador não atua no conjunto dos inteiros. O que acontece na prática é que as pessoas quando estão aprendendo a programar trazem consigo a lógica matemática para escrever seus respectivos códigos. Por isto, tendem a deduzir que todas as operações vistas no conjunto dos reais podem ser aplicadas perfeitamente no contexto de código. Porém, isto não acontece porque as operações computacionais são executadas em outra estrutura algébrica, que não estão contidas todas as propriedades vistas no conjunto dos números $\mathbb{R}$.

3.1 Média Aritmética

Para aprender um novo idioma os professores iniciam suas aulas com exemplos acessíveis para facilitar o entendimento do aluno e, à medida que o aluno entende aquele conceito, as aulas vão aumentando a complexidade.

De forma análoga, isto ocorre também no processo de ensino-aprendizagem de programação. Quando o código compila, o aluno tende a entender que deu tudo certo, deixando passar erros semânticos que podem gerar um conjunto de resultados

inválidos, dependendo do contexto do programa.

No caso de média aritmética, trata-se da "divisão em partes iguais do todo entre seus componentes. Assim, seu algoritmo consiste em somar todos os valores da variável e dividir pelo número de dados"(CAZORLA, 2006). Além disso, sua facilidade de aplicação faz com que ela seja recorrente no dia-a-dia, sendo usada para fornecer critérios de avaliação nas escolas, análises de dados em pesquisas no âmbito acadêmico, crescimento populacional, taxas de rendimento em investimentos e uso de *data science* (MARQUES; GUIMARÃES; GITIRANA, 2011).

Definição 12 (Média Aritmética $\bar{x}$) A média de um conjunto de valores é igual a soma de todos seus elementos dividido pela quantidade de elementos. Em outros termos, a média dos elementos $\{a_1, a_2, \cdots, a_n\}$ é definida por

$$\bar{x} = \frac{a_1 + a_2 + ... + a_n}{n} \tag{3.1}$$

Como foi dito anteriormente, uma aplicação possível para a média aritmética acontece é a média de notas de um aluno como mostra o código 3.1.

Código 3.1 – Exemplo de média aritmética com *overflow*

```
1 media_final = (nota1 + nota2) / 2 ;
2 printf ("%d" , media_final);
```

No exemplo 3.1, é mostrado uma implementação descuidada, porém muito comum, da média aritmética de dois elementos, que pode causar resultados incorretos para um subconjunto significativo de entradas.

Nesta implementação há retorno de valores incorretos sempre que a soma $nota1 + nota2$ resulta em *overflow*, isto é, quando a soma é maior que $2^{31} - 1$, que é o maior valor representável por variáveis inteiras em arquiteturas de 32 *bits*.

Supondo duas variáveis genéricas a e b, onde $a = nota1$ e $b = nota2$.
Se $a \leq b < 2^{32}$, então a média m também caberá um *int*, pois $a \leq m \leq b$

$$m = \frac{a+b}{2} = \frac{a}{2} + \frac{b}{2} \leq \frac{b}{2} + \frac{b}{2} = b$$

Assim, a média pode ser reescrita como

$$m = \frac{a+b}{2} = \frac{a}{2} + \frac{b}{2} = \left(a - \frac{a}{2}\right) + \frac{b}{2} = a + \left(\frac{b-a}{2}\right)$$

Dessa forma, para solucionar este problema, modifica-se a implementação da média aritmética para os elementos *nota1* e *nota2* da forma apresentada no Código 3.2.

Código 3.2 – Exemplo de média aritmética sem *overflow*

```
1
2    int mean_no_overflow(int nota1, int nota2) {
3        return nota1 + (nota2 - nota1)/2;
4    }
```

A função de média implementada recebe dois valores a e b, sendo o primeiro deles sempre menor que o segundo. Para exemplificar a importância dessa pré condição, suponha $a = 4$ e $b = 7$, eles retornam média igual a 5 para a função com *overflow* enquanto para o segundo caso retorna valor 6. E este, claramente não é um problema de *overflow* mas sim um falso positivo. Já na situação da função média receber respectivamente 7 e 4, as duas médias retornam 5.
A Tabela 4 ilustra alguns exemplos de cálculos de médias aritméticas em computador, que foram gerados randomicamente em C++, com ambas expressões.

.

Tabela 4 – Exemplo de equações para cálculo da média

a	b	$\frac{a+b}{2}$	$a + \frac{b-a}{2}$
2147483647	1	-1073741824	1073741824
2147483647	2147483647	-1	2147483647
1073741824	1073741824	-1073741824	1073741824

Observe que a terceira linha da Tabela 4 apresenta o mesmo valor para as duas primeiras colunas, ou seja $a = b = 1073741824$. E para este caso espera-se um retorno igual à a ou b para a média aritmética entre eles, porém quando a média aritmética é calculada guiada pela equação $\frac{a+b}{2}$, referente à terceira coluna ocorre *overflow*.

Para determinar a quantidade de casos que há divergencia no retorno da média, ou seja, encontrar a porcentagem de erro no código de média aritmética será apresentada a respectiva demonstração matemática.
Dados dois valores inteiros a e b, no intervalo onde $a, b \in [0, 2^{32} - 1]$ como mostra o Código 3.3.

Código 3.3 – Exemplo de média aritmética sem *overflow*

```
1    for a in range(0, 2^(32)-1:
2        for b in (2^(32)-a, 2^(32))
```

O cálculo da média aritmética dispõe um total de pares(a, b) de $2^{32} \cdot 2^{32} = 2^{64}$, isto é, 2^{64} entradas distintas. Para determinar quantos desses pares levam ao *overflow*, calcula-se:

$$\begin{aligned} T &= \sum_{a=0}^{2^{32}-1} \sum_{b=2^{32}-a}^{2^{32}-1} 1 \\ &= \sum_{a=0}^{2^{32}-1} \sum_{b=a}^{2^{32}-1} a = \frac{(2^{32}-1) \cdot (1 + 2^{32} - 1)}{2} \end{aligned} \tag{3.2}$$

Esta equação pode ser reescrita como a multiplicação do número de elementos pela soma do primeiro elemento com o último, veja:

$$\begin{aligned} T &= \frac{(2^{32}-1) \cdot (1 + 2^{32} - 1)}{2} \\ &= \frac{(2^{32}-1) \cdot 2^{32}}{2} \\ &= (2^{32}-1) \cdot 2^{31} \end{aligned} \tag{3.3}$$

Para encontrar a probabilidade da média, isto é, resultar *overflow* calculamos o

número de casos desejados dividido pelos casos possíveis:

$$\begin{aligned} P(a,b) =& \frac{2^{31} \cdot (2^{32}-1)}{2^{32} \cdot 2^{32}} \\ &= \frac{1}{2} \cdot \frac{2^{32}-1}{2^{32}} \\ &= \frac{1}{2} - \frac{1}{2 \cdot 2^{32}} \\ &= \frac{1}{2} - \frac{1}{2^{33}} \approx 50\% \end{aligned} \tag{3.4}$$

A demonstração matemática mostra que 50% dos pares de elementos levam a resultados incorretos ao calcular a média aritmética entre dois elementos utilizando o código com *overflow*. O próximo passo é fazer um experimento que compara os códigos com presença e ausência de *overflow* para verificar que a taxa de erro se confirma na prática utilizando pares a e b aleatórios.
E para fazer essa comparação foram testados valores randômicos para duas variáveis a e b. Em seguida, comparando o retorno das funções 3.1 e 3.2, e incrementando em `errors` quando houvesse resultados distintos.

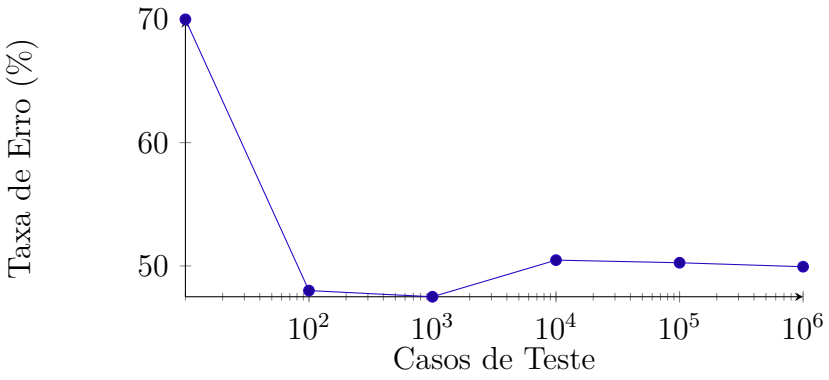

Figura 5 – Experimento referente à Média Aritmética

Portanto, os pares ordenados com um milhão de amostras apresentados no Gráfico pela Fugura 5 evidencia que a taxa de erro converge para 50% de erros causados pelo *overflow*.

3.2 Mínimo Múltiplo Comum

Assim como no caso de média aritmética, para encontrar o menor divisor comum entre dois números em código, ele pode retornar valores inválidos causados pelo *overflow*, dependendo das entradas e como esse código foi desenvolvido.
Conforme apresentado na Definição 3, podemos encontrar o MMC multiplicando as entradas a e b, e dividindo-os pelo maior múltiplo comum (mdc) entre eles, como mostra o Código

```
1    int mdc(int a, int b){
2        while(b != 0){
3            int r = a % b;
4            a = b;
5            b = r;
6        }
7        return a;
8    }
9
10   int mmc_overflow(int a, int b) {$$
11       return (a * b) / MDC(a,b);
12   }
```

Código 3.4 – Exemplo de média aritmética com *overflow*

Como mostra o código 3.4, o maior divisor comum, é calculado por divisões sucessivas de dois valores a e b sendo $a \geq b$. Por exemplo o MDC (6,12) = 6, porque 6 é o maior número que divide 6 e também divide 12. O algoritmo de Euclides 2 utilizado na implementação desta função restringe o intervalo para valores positivos. Com esta pré condição estabelecida não é possível existir *overflow*.
Já no caso da função seguinte que calcula o mínimo múltiplo comum, como mostra o código 3.4 ambas as variáveis a e b tem uma capacidade de armazenamento restrita a $2^{32} - 1$, valor disponível para os inteiros sem *bit* de sinal. E o cálculo do mmc é realizado pela multiplicação desses dois parâmetros a e b e depois dividido pelo retorno do mdc. Porém, se o valor de a for o maior número que uma variável do tipo inteira é capaz de armazenar, então para qualquer valor acima de 1 ocorrerá *overflow*.
De maneira geral, o *overflow* acontece sempre que o produto ab ultrapassa $2^{31} - 1$.

Outro exemplo é o caso onde a e b são ambos iguais a 2^{16}, de maneira que o resultado da multiplição é 2^{32}, que é igual a zero no anel $\mathbb{Z}_{2^{32}}$.
Nestes casos, no código 3.4, apenas depois de estourar a capacidade de armazenamento, ou seja quando há *overflow*, este resultado é dividido pelo MDC. Com intuito de evitar um resultado incorreto o ideal é fazer essa divisão antes do *overflow* como mostra o Código 3.5.

```
1 int mmc_no_overflow(int a, int b)
2 {
3     return (a/mdc(a,b))*b;
4 }
```

Código 3.5 – Exemplo de média aritmética com *overflow*

Para resolver este problema, foi realizado um experimento que gera a taxa de erro ao comparar o trecho de código com presença e ausência de *overflow* para 10 até 10^6 casos de testes com valores randomicos de a e b a fim de em encontrar quantos trios (a, b, MDC) apresentam resultados inválidos.
Um exemplo gerado randomicamente neste experimento foi `2116740715` para a e `920997115` para b, que resultou no mínimo múltiplo comum equivalente a `107986069`. Porém este valor não está correto, visto que o resultado deveria gerar um número maior ou igual ao maior deles. Na verdade, o MMC correto entre eles é `1949512087113051 65`.
Para corrigir esse problema, foi codificado a pré condição de $a \geq b$ resultado realmente é um número múltiplo dos elementos em questão conforme ilustra a Figura 6.

A Figura 6 mostra a comparação entre valores randômicos para duas variáveis a e b. Pra tal, foram calculados os respectivos MMC com e sem *overflow* e então foram geradas as porcentagens de erros para seis casos de testes: a média de 10 pares escolhidos aleatoriamente no intervalo $[1, k]$, onde k é uma potência de 10 menor ou igual a 10^6. Este problema se fez necessário porque há chance de estourar a capacidade de armazenamento do produto ab. Por exemplo, se $a = 2^{16}$ e $b = 2^{16}$, sendo que a variável do tipo int armazena $2^{32} - 1$, apesar das variáveis isoladas armazenarem seus valores, o produto ab gera *overflow*. Assim como esse exemplo, 31 % dos casos gera valores incorretos causados pelo *overflow*.

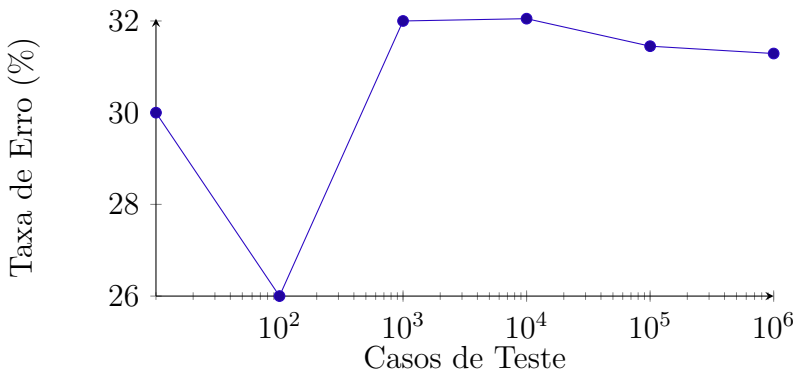

Figura 6 – Experimento referente ao Mínimo Múltiplo Comum

3.3 Resto da divisão

A divisão euclidiana é ensinada para crianças associadas, por exemplo, ao número de balinhas que devem ser repartidas entre elas, e assim elas aprendem que em divisões não exatas há balinhas restantes, isto é, o resto da divisão é um número inteiro maior que zero. Este cálculo é reforçado durante o ensino fundamental e médio, com outras contextualizações como a divisão de polinômios.
Pelo Teorema 2, um número inteiro pode ser reescrito como $a = bq+r$ com $0 \leqslant r < |b|$. De maneira análoga, um polinômio pode ser escrito como $p(x) = (x-a)q(x)+r$, se $x = a$ então: $p(a) = (a-a)q(a)+r$, isto é, $p(a) = 0+r$ assim concluimos que $p(a) = r$. Assim como essas aplicações matemáticas, no contexto de programação existem numerosas aplicações para o cálculo do resto da divisão e para elas usa-se o operador `mod`, em C/C++ escreve-se %. Por exemplo, na implementação de uma calculadora em C++, como mostra o trecho de Código[1] 3.6:

```
1   int num1, num2, resto;
2   resto = (num1% num2);
```

Código 3.6 – Exemplo de resto da divisão

Assim como este exemplo, a maior parte dos códigos que calculam o resto da divisão utilizam o operador `mod`, disponível pelo código-fonte de cada linguagem. Em C++, por exemplo, esse operador é implementado pelo símbolo %. De maneira geral, os programadores não costumam consultar a documentação para certificar como foi

[1] https://github.com/Carvalhogyn001/Calculadora/blob/master/Calculadora.cpp

implementada cada operação utilizada no dia-a-dia e muitas vezes presumem que o resto da divisão é calculado como eles aprederam na escola e mostra o Teorema 2.

Porém, o cálculo de Euclides para resto da divisão restringe o resto ao intervalo $[0, b)$, e este intervalo efetivamente corresponde a um sistema completo de restos. Mas o algoritmo pode ser modificado para que o resto fique restrito a outros sitemas de restos, como o sistema de menor resto, cujos representantes são os números $\frac{-b}{2}, \cdots, 0, 1, 2, \frac{b}{2}$. Isto significa que dependendo da linguagem escolhida um mesmo código usa algoritmos diferentes para calcular uma divisão resultando em restos distintos.

Suponha que se queira o resto da divisão de 5 por 3 e para isso utilize o *Python*, o resultado será 2, pois $5 = 1(3) + 2$. Se repetimos o processo, calculando a divisão de -7 por 3, o resultado será 2, pois $-7 = 3(-3) + 2$, ou seja o resto resultou em um valor positivo. Já quando se quer esse mesmo resultado em outra linguagem como C++, o primeiro cálculo com o numerador positivo apresenta o mesmo resultado nas duas linguagens, enquanto com numerador negativo apresenta um valor diferente, no caso equivalente a -1, pois $-7 = 3(-2) - 1$, contudo $-1 \equiv 2 \pmod 3$.

Até o momento, entende-se que este problema refere-se a utilizar termos iguais para conceitos matemáticos distintos, e assim induzir o programador a cometer erros semânticos desnecessários. Para evitar este problema, será apresentada uma classificação de diferentes linguagens, segundo o desempenho delas em calcular o resto da divisão, ou seja, se seus respectivos operadores utilizam no código-fonte o algoritmo de Euclides ou o sistema de menor resto.

Para definir as linguagens que foram implementadas o cálculo do menor resto, e então clasificá-las, consultou-se o índice Tiobe[2] que classifica as linguagens mais populares mensalmente. Para dizer que uma linguagem é realmente popular são analisadas caracteríticas da páginas nas buscas do *Google, Bing, Yahoo !, Wikipedia, Amazon*, por exemplo se a página acessada tem barra de pesquisa, ou indicação do número de acesso à página, ou ainda, se os resultados da página que referencia uma linguagem de programação estão disponíveis em HTML.

O resultado do índice TIOBE, em abril de 2019, diz que as nove linguagens mais populares são: Java, C, Python, C++, Visual Basic.net, C#, Javascript e PHP. E

[2] https://www.tiobe.com/tiobe-index/

então foi implementado o resto da divisão em cada uma delas, no entanto verificou-se que apenas a linguagem *Python* se comportava segundo a divisão de menor resto. A fim mostrar outros exemplos com resto da divisão implementada utilizando o sistema de menor resto, esta implementação foi realizada também nas linguagens: Fortran, Perl e Ruby. Por fim, a Tabela 5 apresenta a classificação das linguagens abordadas neste texto, e o Anexo C apresenta as respectivas implementações.

Tabela 5 – Classificação de algoritmos de Euclides ou Menor resto.

Linguagem	**Euclides**	**Menor Resto**
Java	•	
C	•	
Python		•
C++	•	
Visual basic.NET	•	
C#	•	
JavaScript	•	
PHP	•	
Fortran		•
Perl		•
Ruby		•

3.4 Divisão por zero

A divisão por zero acontece sempre que o denominador é zero. Em cálculo é possível utilizar limite para manipular as equações e apresentar informações sobre aqueles polinômios e seus respectivos gráficos. Já no computador não estamos trabalhando com o conjunto dos números inteiros, na verdade o computador usa estruturas matemáticas mais elaboradas como o anel, visto no Capítulo 1.

Como já sabemos o anel tem a propriedade de congruência, o que significa que em uma arquitetura de 64 *bits*, dados a e b inteiros, vale: $a \equiv b \pmod{2^{64}}$. Isto significa que em uma equação genérica $x = \frac{k}{a*b}$, onde a multiplicação $a * b$ é congruente a zero sempre que seu resultado é múltiplo a 2^{64}, ou seja, como essa multiplicação está no denominador, existe uma divisão por zero nesta situação.

Por outro lado, se $a = 2$ e $b = 0$ apesar do resultado ser uma indeterminação o computador entende como: calcular a multiplicação de $\frac{k}{a}$ por b, o que significa que esse cálculo geram resultados inválidos como mostra o Código 3.7

Código 3.7 – Exemplo da divisão por zero com *overflow*

```
1    int divisao_zero_overflow(int a, int b){
2        int const K {1000000000};
3
4        if(a == 0 || b== 0){
5            return -1;
6        }
7        return K/(a*b);
8    }
```

O problema dessa implementação é que existem pares de valores $a*b$ que não foram incluidos no conjunto de valores inválidos, como por exemplo $a = 2^{32}$ e $b = 2^{32}$ ou seja o produto ab resulta em 2^{64} que é congruente a zero, tornando esse exemplo uma divisão por zero. Para resolver este problema foi implementada uma alternativa ao código inicial como mostra o Código 3.8

Código 3.8 – Exemplo da divisão por zero sem *overflow*

```
1    int divisao_zero_sem_overflow(int a, int b){
2        int const K {1000000000};
3
4        if(a * b == 0){
5            return -1;
6        }
7        return K/(a*b);
8    }
```

Um exemplo prático onde esse problema acontece é no cálculo do índice de massa corporal(IMC), calculado por $IMC = \frac{massa}{altura^2}$. O Código 3.8 mostra um exemplo da divisão de números inteiros, a qual tem restrições no seu cálculo. Por exemplo, se denominador for zero, ocorre um erro de execução. Já no caso de números decimais, variáveis do tipo `float`, essa divisão é tratada usando limites máximo e mínimo. Algumas dessas diferenças entre números inteiros e decimais (ponto flutuante) estão mostradas na Tabela 6.

.

Tabela 6 – Exemplo de overflow em divisão por zero

Inteiro	**Ponto Flutuante**
Intruções Primitivas (exceto %)	Coprocessador ou implementação de *software*.
Resultado exato	Resultado aproximado
Não permite divisão por zero	Divisão por zero tratada no limite $+\infty$ e $-\infty$

Como foi dito anteriormente, o computador não executa suas operações no conjunto dos números inteiros. Na verdade ele atua no anel. Assim, existem diferenças entre a aritmética no conjunto dos números inteiros e aritmética que os computadores usam em suas representações de inteiros.

Estes anéis finitos, e as operações computacionais são calculadas com as propriedades da aritmética modular, como a congruência. Supondo a arquitetura do computador requerido seja de 2^{32} entende-se que o valor máximo de uma operação é $2^{32}-1$, o que significa que $2^{32} \equiv 0 (\text{mod } 2^{32})$. Se houver uma equação cujo denominador seja 2^{32}, este caso retornará um erro por se tratar de divisão por zero.

Existem produtos ab congruentes a zero, onde ambos a e b são não-nulos, o que leva a divisões por zero que puderam passar despercebidas. Para evitá-los este estudo, apresenta o intervalo de valores que não retornam resultados errôneos.

O cálculo do IMC utiliza a massa corporal (m) em gramas e a altura (h) em centímetros. Ele é definido por:

$$IMC = \frac{m}{h^2}$$

Supondo que o IMC está sendo realizado em um computador de 32 *bits* conforme apresentado no Problema 3.4, e as variáveis m e h estejam nos intervalos: $m \in [1, M]$ e $h \in [1, H]$.

Existe o total de MH pares de entradas possíveis. Os erros aconteccem quando $h^2 = 2^{32}$ ou $h^2 = 2^{64}$ dependendo da arquitetura. Supondo uma arquitetura de 32 *bits* e $h = 2^{16}$, então, se $H \geqslant 2^{16}$, neste contexto existem M pares com resultads errados, e portanto taxa de $\frac{1}{H}$ de erro, porque existe um único evento desejado que é o valor h $= 2^{16}$ em um contexto de H possibilidades. Enquanto no caso $H < 2^{16}$, todos os resultados são validados como mostra a Figura 7.

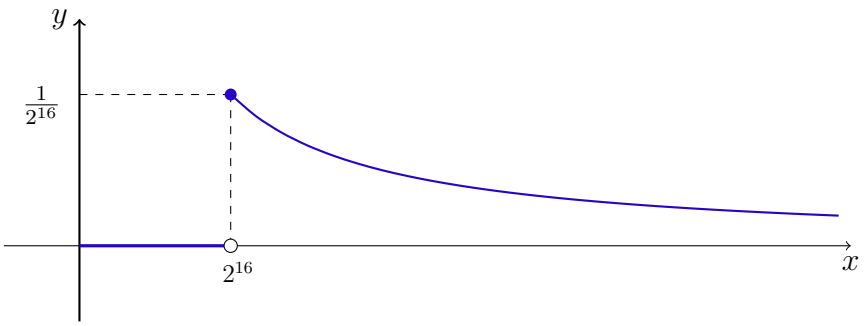

Figura 7 – Valores válidos para equações cujo denominador é um produto.

Neste exemplo a divisão por zero acontece quando o denominador é 2^{16}, ou simplesmente 65356.
Apesar de serem poucos valores sucetíveis ao erro, existe possibilidade do código quebrar ou ser *hackeado* por isso a importância de produzir *softwares* seguros e minimizar os riscos. Prinicpalmente por se tratar de uma situação típica, como o uso de frações nas implementações.

3.5 Associatividade

A soma ou multiplicação de dois elementos sempre resulta em valores iguais, por conta da propriedade comutativa destas operações. Quando essas operações são acrescidas de um ou mais elementos, espera-se resultados equivalêntes por conta da propriedade de associatividade.
Conforme já viamos abordando nos problemas anteriores, o computador não trabalha no conjunto dos números inteiros mas sim como anéis e portanto nem todas as propriedades do conjunto dos números inteiros estão presentes como por exemplo a associatividade.
Especificamente, a associatividade não se mantém quando trabalhamos com números em ponto flutuante (*float*). Isso significa que o mesmo programa retorna valores diferentes dependendo da associação utilizada na adição ou multiplicação dos seus elementos.
As variáveis em ponto flutuantes podem ser armazenadas em dois tipos: precisão simples e precisão dupla (`float` e `double`). Variáveis com precisão simples usam 32

bits, sendo o primeiro *bit* usado para representar o sinal, em seguida são reservados 8 *bits* para o expoente, e por fim, 23 *bits* para a parte fracionária, conforme mostra a Figura 8 (STEVENSON, 1981).

Figura 8 – Formato de ponto flutuante em precisão simples (32 bits)

Para a precisão simples existem cinco casos pela norma 754 (STEVENSON, 1981) que definem o valor de um número:

1. Se $e = 255$ e $f \neq 0$ então o valor v = NaN (*not a number*)
2. Se $e = 255$ e $f = 0$, então v = $(-1)^s \infty$
3. Se $0 > e < 255$ então v = $(-1)^s \cdot (1.f) \cdot 2^{e-127}$
4. Se $e = 0$ e $f \neq 0$ então v = $(-1)^s \cdot (0.f) \cdot 2^{-126}$
5. Se $e = 0$ e $f = 0$ então v = $(-1)^s \cdot (0)$ (zero)

Por exemplo o número $0\_10000001010000000000000000000000_{(2)}$ se encaixa no caso 3 porque o valor de e está entre 0 e 255. Utilizando a fórmula do caso 3, temos que o valor deste número equivale a: $v = ((-1)^0 \cdot (1.01) \cdot 2^{129-127})_2 = (1.01 \cdot 2^2)_2 = 5_{10}$ De maneira análoga, para números em precisão dupla são representados usando um *bit* de sinal, 11 *bits* para o expoente e 54 *bits* para a parte fracionária como mostra a Figura 9.

Figura 9 – Formato de de ponto flutuante precisão larga (64 bits)

Para a precisão dupla são cinco casos que definem o valor de um número, são eles:

1. Se $e = 2047$ e $f \neq 0$ então o valor v = NaN (*not a number*)
2. Se $e = 2047$ e $f = 0$ então v = $(-1)^s \infty$

3. Se $0 > e < 2047$ então v $= (-1)^s \cdot (1.f) \cdot 2^{e-1023}$
4. Se $e = 0$ e $f \neq 0$ então v $= (-1)^s \cdot (0.f) \cdot 2^{-1022}$
5. Se $e = 0$ e $f = 0$ então v $= (-1)^s \cdot (0)$ (zero)

Para identificar o valor de um número basta seguir identificar se o número é de 32 ou 64 *bits*, identificar os valores de e e f e aplicar as condições mostradas neste trabalho e propostas pela norma 754 (STEVENSON, 1981).
Portanto, a soma de três ou mais números em ponto flutuante retorna resultados diferentes dependendo da ordem que eles são dispostos no código. Por exemplo, $a = -8.363287925720215$, $b = 7.233823299407959$, $c = -0.555755972862244$. A operação $(a + b) + c$ resulta em -1.68522059917449951172 enquanto a operação $a + (b + c)$ resulta em -1.68522071838378906250.
Conforme descrito no Problema 3.5, a associatividade não se mantém nas operações de adição e multiplicação entre três elementos em ponto flutuante. Supondo três números em ponto flutuantes: $a = -8.363287925720215$, $b = 7.233823299407959$, $c = -0.555755972862244$. A operação $(a + b) + c$ resulta em -1.68522059917449951172 enquanto a operação $a + (b + c)$ resulta em -1.68522071838378906250.
De maneira análoga, os números a=5.403974056243896, b=-9.246410369873047 e c=-6.739072799682617 quando multiplicados no formato $(ab)c$ resulta em 336.733673095 enquanto os mesmo números multiplicados no formato $a(bc)$ resulta em 336.733703613.
Para evidenciar as taxas de erro na adição de três elementos em ponto flutuante foi feito um experimento em C++ que compara retorno dessa soma alterando a ordem dos fatores, conforme apresenta a Figura ??.
Portanto, nesta amostra cujo intervalo é de $[1, 10^6]$ elementos, a soma de três números aleatórios em ponto flutuante retorna resultados errados em $36,1\%$ dos casos e na multiplicação essa taxa aumenta para 38.8% dos casos. Esse resultado mostra que a ordem das operações podem modificar os resultados.

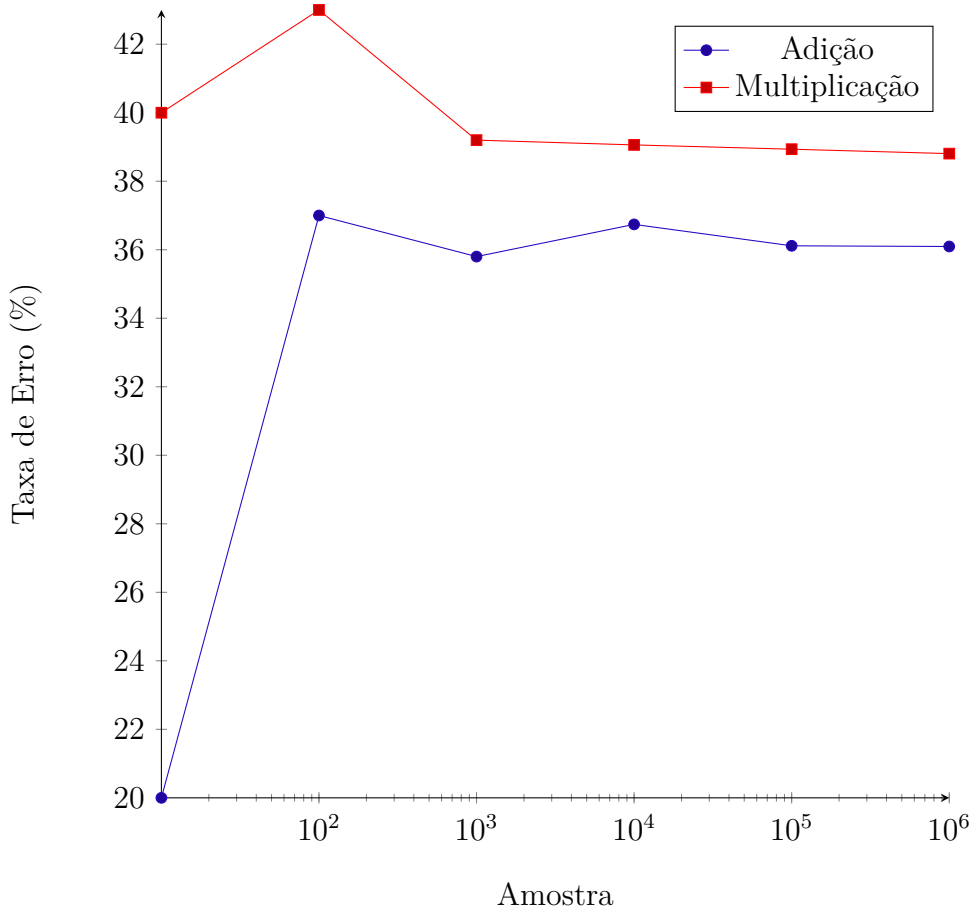

Figura 10 – Experimento referente a associatividade

4 Considerações Finais

Dada a importância de produzir códigos com resultados que evitem erros semânticos, um contexto onde não tem-se bibliográfia de como evitar tais erros, uma categorização se faz necessária. Este estudo pode auxiliar programadores a produzirem implementações que gerem resultados corretos.

Este trabalho demandou um conhecimento matemático não abordado no curso de Engenharia de *Software*, de maneira geral existe uma densidade conceitual envolvido nos problemas e erros, outra dificuldade foi a dificuldade em encontrar problemas com erros semânticos, visto que não foi encontrado material bibliográfico que relate o assunto nem há bases de erros catalogados com soluções para estes problemas.

Por fim, como foi mostrado neste trabalho foram realizadas listagem de cinco problemas que geram resultados inválidos, feito as demonstrações matemáticas para os casos cabíveis, os experimentos via código gerando valores randômicos para testar código com valores errôneos e valores corretos de acordo com a distribuição gaussiana, e por fim a plotagem desses valores no gráfico e sua respectiva discussão.

Referências

AROEIRA, A. L. K. Algoritmos para multiplicação rápida de dois números com mais de 1.000 dígitos cada. 2018. Citado na página 16.

CAZORLA, I. M. *Metodologia do ensino da matemática*. [S.l.]: UESC, 2006. 18 p. Citado na página 32.

DRUGAN, J. Top-down or bottom-up: what do industry approaches to translation quality mean for effective integration of standards and tools? *University of East Anglia*, 2014. Citado na página 28.

EVARISTO, J.; PERDIGÃO, E. *Introdução à Algebra Abstrata*. [S.l.]: Editora da Universidade Federal de Alagoas, 2002. 49-56 p. Citado na página 22.

FLOYD, T. *Sistemas Digitais*: Fundamentos e aplicações. [S.l.]: Bookman, 2007. Citado na página 21.

GONCALVES, A. *Introdução à álgebra*. [S.l.]: IMPA, 2002. 28 p. Citado na página 22.

GUIDETTI, S. A. et al. Aplicação de analise de mutantes a geração de dados de teste para detecção de vulnerabilidade do tipo buffer overflow. [sn], 2005. Citado na página 21.

IEZZI, H. H. D. G. *Algebra Moderna*. [S.l.]: Saraiva S.A. Livreiros Editores, 2003. 218-231 p. Citado na página 23.

LAMPORT, L. *LATEX: a document preparation system: user's guide and reference manual*. [S.l.]: Addison-wesley, 1994. Citado na página 30.

LIEDL, M. R. C. Does a top-down approach bear more advantages than a bottom-up approach within the implementation process of housing security projects? *University Twente*, 2011. Citado na página 28.

MARQUES, M.; GUIMARÃES, G. L.; GITIRANA, V. Compreensões de alunos e professores sobre média aritmética. *Bolema-Boletim de Educação Matemática*, v. 24, n. 40, p. 725–745, 2011. Citado na página 32.

NEVES, V. Introdução à teoria dos números. 2011. Citado na página 16.

RAUPP, F. M.; BEUREN, I. M. Metodologia da pesquisa aplicável às ciências. *Como elaborar trabalhos monográficos em contabilidade: teoria e prática. São Paulo: Atlas*, 2006. Citado na página 25.

RITCHIE, D. M. The development of the c programming language. In: ACM. *History of Programming languages—II*. [S.l.], 1996. p. 671–698. Citado na página 21.

ROSSA, L. P. et al. Indeterminação semântica: ambigüidade, vagueza e polissemia na teoria da relevância. Florianópolis, SC, 2001. Citado na página 14.

STEVENSON, D. A proposed standard for binary floating-point arithmetic. *Computer*, IEEE, n. 3, p. 51–62, 1981. Citado 2 vezes nas páginas 44 e 45.

Anexos

ANEXO A – Anexo: Média Aritmética

```
1     #include <bits/stdc++.h>
2
3  using namespace std;
4  using ii = pair<int, int>;
5
6  int mean_overflow(int a, int b)
7  {
8      return (a + b)/2;
9  }
10
11 int mean_no_overflow(int a, int b)
12 {
13     int c, d;
14     if(b>a){
15         c=a;
16         d=b;
17     }else{
18         c=b;
19         d=a;
20     }
21     return c + (d - c)/2;
22 }
23
24 double experiment(int N)
25 {
26     vector<ii> samples(N);
27     srand(time(NULL));
28
29     for (int i = 0; i < N; ++i)
30     {
31         samples[i].first = rand();
32         samples[i].second = rand();
33     }
34
35     int errors = 0;
```

```
36
37      for (const auto& p : samples)
38      {
39          auto a = p.first;
40          auto b = p.second;
41
42          auto x = mean_overflow(a, b);
43          auto y = mean_no_overflow(a, b);
44
45          if (x != y)
46              ++errors;
47      }
48
49      return (double) errors / N;
50  }
51
52  int main()
53  {
54
55      vector<int> ns { 10, 100, 1000, 10000, 100000, 1000000 };
56
57      ofstream arq;
58      arq.open("media.csv");
59
60      arq << "N" << ";" << "Erro" << endl;
61
62      for (auto N : ns)
63      {
64          cout << "N = " << N << '\n';
65          cout << "error = "  << experiment(N) << '\n';
66          arq << N << ";" << experiment(N) << endl;
67      }
68      return 0;
69  }
```

ANEXO B – Anexo: Mínimo Múltiplo Comum

```
#include <bits/stdc++.h>

using namespace std;
using ll = pair< int, int>;

int mdc(long long a, long long b){
    while(b != 0){
        int r = a % b;
        a = b;
        b = r;
    }
    return a;
}

int mmc_overflow(long long a, long long b)
{
    return (a*b)/mdc(a,b);
}

int mmc_no_overflow(long long a, long long b)
{
    long long c =a/mdc(a,b);
    c= c*b;
    return c;
}

bool divisible(int x, int y){
    int z;
    if(y==0){
        return false;
    }
    z= x/y;

```

```
34          z= z *y;
35
36          if(z==x){
37              return true;
38          }else
39          {
40              return false;
41          }
42      }
43
44      double experiment(int N)
45      {
46          vector<ll> samples(N);
47          srand(time(NULL));
48
49          for (int i = 0; i < N; ++i)
50          {
51              samples[i].first = rand()%100000;
52              samples[i].second = rand()%100000;
53          }
54
55          int errors = 0;
56
57          for (const auto& p : samples)
58          {
59              auto a = p.first;
60              auto b = p.second;
61
62              auto x = mmc_overflow(a, b);
63              auto y = mmc_no_overflow(a, b);
64
65              if (x != y){
66                  ++errors;
67              }else{
68                 if(!divisible(x,a) ||!divisible(x,b)){
69                    //cout << a << " "<< b<< " "<< x << endl;
70                    ++errors;
71                 }
72              }
```

```
73
74          }
75
76          return (double) errors / N;
77      }
78
79      int main()
80      {
81      /*      vector<ii> overflow_examples { ii(2147483647, 1), ii
    (2147483647, 2147483647), ii(1073741824, 1073741824) };
82
83          for (const auto& p : overflow_examples)
84          {
85              auto a = p.first;
86              auto b = p.second;
87
88              cout << "overflow = " << mean_overflow(a, b) << '\n';
89              cout << "no overflow = " << mean_no_overflow(a, b) <<
    '\n';
90          }
91      */
92
93          vector<long long> ns { 10, 100, 1000, 10000, 100000,
    1000000 };
94
95          ofstream arq;
96          arq.open("mmc.csv");
97
98          arq << "N" << ";" << "Erro" << endl;
99
100         for (auto N : ns)
101         {
102             cout << "N = " << N << '\n';
103             cout << "error = "  << experiment(N) << '\n';
104             arq << N << ";" << experiment(N) << endl;
105         }
106
107         return 0;
108     }
```

ANEXO C – Anexo: Resto da Divisão

C.1 Java

```java
public class Main{
    public static void main(String[] args) {
        int i = 0;
        i = (-7 % 3);
        System.out.println(i);
    }
}
```

C.2 C

```c
#include <stdio.h>
int main(){
    int a = -7%3;
    printf("%d",a);

    return 0;
}
```

C.3 Python

```python
a= -7%3
print(a)
```

C.4 C++

```cpp
#include <iostream>
#include <string>

int main(){
int a= -7%3;
std::cout << a ;
}
```

C.5 Visual basic.NET

```
Imports System
Public Class Test
    Public Shared Sub Main()
        Dim x, y as Integer
        x = -7 Mod 3
        System.Console.WriteLine("x " & x)

    End Sub
End Class
```

C.6 C#

```
using System;
public class Program
{
    public static void Main()
    {
        Console.WriteLine(-7 % 3);
    }
}
```

C.7 Javascript

```
print(-7 % 3)
```

C.8 PHP

```
<?php
    echo (-7 % 3);
?>
```

C.9 Ruby

```
print -7%3
```

C.10 Perl

```
print -7%3
```

C.11 Fortran

```
program test_modulo
    print *, modulo(-7,3)
end program
```